# Inheritance and Polymorphism

- Inheritance (Continued)
- Polymorphism
- Polymorphism by inheritance
- Polymorphism by interfaces
- Reading for this lecture: L&L 10.1 – 10.3

# Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- That is, one interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the ancestor and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

# Visibility Revisited

- All variables and methods of a parent class, even private members, are inherited by its children
- As we've mentioned, private members cannot be referenced by name in the child class
- However, private members inherited by child classes exist and can be referenced indirectly
- Because the parent can refer to the private member, the child can reference it indirectly using its parent's methods
- The `super` reference can be used to refer to the parent class, even if no object of the parent class exists

# Designing for Inheritance

- As we've discussed, taking the time to create a good software design reaps long-term benefits
- Inheritance issues are an important part of an object-oriented design
- Properly designed inheritance relationships can contribute greatly to the elegance, maintainability, and reuse of the software
- Let's summarize some of the issues regarding inheritance that relate to a good software design

# Inheritance Design Issues

- Every derivation should be an is-a relationship
- Think about a potential future class hierarchy
- Design classes to be reusable and flexible
- Find common characteristics of classes and push them as high in the class hierarchy as appropriate, i.e. “generalize” the behavior
- Override methods as appropriate to tailor or change the functionality of a child
- Add new variables to children, but don't redefine (shadow) inherited variables

# Inheritance Design Issues

- Allow each class to manage its own data; use the `super` reference to invoke the parent's constructor to set up its data
- Even if there are no current uses for them, override general methods such as `toString` and `equals` with appropriate definitions
- Use abstract classes to represent general concepts that lower classes have in common
- Use visibility modifiers carefully to provide needed access without violating encapsulation

# Restricting Inheritance

- The `final` modifier can be used to curtail inheritance
- If the `final` modifier is applied to a method, then that method cannot be overridden in any descendent classes
- If the `final` modifier is applied to an entire class, then that class cannot be used to derive any children at all
  - Thus, an abstract class cannot be declared as final
- These are key design decisions and establish that a method or class must be used “as is” or not at all

# Polymorphism

- The term *polymorphism* literally means "having many forms"
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- All object references in Java are potentially polymorphic and can refer to an object of any type compatible with its defined type
- Compatibility of class types can be based on either Inheritance or Interfaces

# Polymorphism

- Suppose we create the following object reference variable (`Holiday` can be a class or an interface):

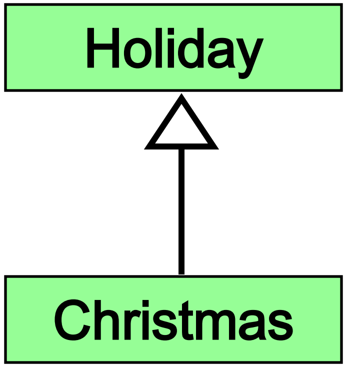
```
Holiday day;
```

- Java allows this reference to point to a `Holiday` object or to any object of any compatible type
- If `class Christmas extends Holiday` or if `class Christmas implements Holiday`, a `Christmas` object is a compatible type with a `Holiday` object and a reference to one can be stored in the reference variable `day`:

```
day = new Christmas();
```

# References and Inheritance

- An object reference can refer to an object of its class or to an object of any class related to it by inheritance
- For example, if the `Christmas` class extends the `Holiday` class, then a `Holiday` reference could be used to point to a `Christmas` object



```
Holiday day;
day = new Christmas();
```

# References and Inheritance

- Assigning a child object to a parent reference is considered to be a widening conversion, and can be performed by simple assignment
- The widening conversion is the most useful
- Assigning a parent object to a child reference can be done, but it is considered a narrowing conversion and two rules/guidelines apply:
  - A narrowing conversion must be done with a cast
  - A narrowing conversion should only be used to restore an object back to its original class (back to what it was “born as” with the new operator)

# Polymorphism via Inheritance

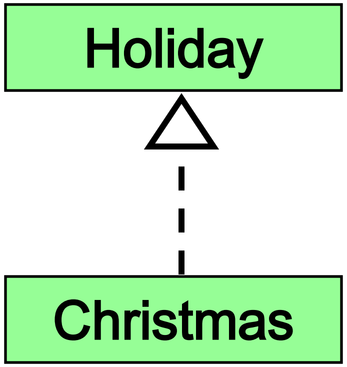
- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- If the `Holiday` class has a `celebrate` method, and the `Christmas` class overrides it, consider the following invocation:

```
day.celebrate();
```

- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate()`
- If `day` refers to a `Christmas` object, it invokes the `Christmas` version of `celebrate()`

# References and Interfaces

- An object reference can refer to an object of its class or to an object of any class related to it by an interface
- For example, if a `Christmas` class implements `Holiday`, then a `Holiday` reference could be used to point to a `Christmas` object



```
Holiday day;
day = new Christmas();
```

# Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable

```
Speaker current;
```

- The `current` reference can be used to point to any object of any class that implements the `Speaker` interface
- The version of `speak` that the following line invokes depends on the type of object that `current` is referencing

```
current.speak();
```

# Polymorphism via Interfaces

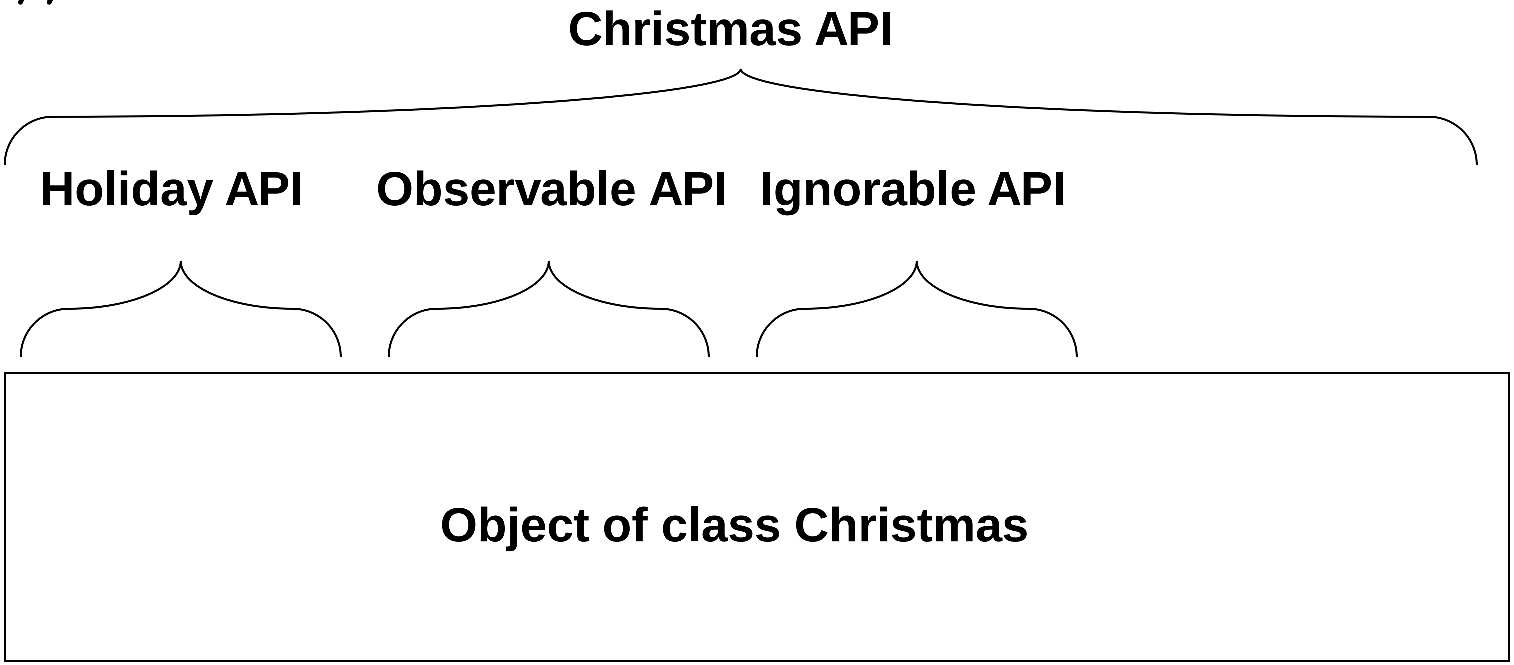
- Suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, but each provides a distinct version of the `speak` method
- In the following code, the first call to `speak` invokes the `Philosopher` method and the second invokes the `Dog` method:

```
Speaker guest = new Philosopher();
guest.speak();  // To be or not to be
guest = new Dog();
guest.speak();  // Arf, Arf
```

# Summary of Polymorphism

Why only one class name here?

```
public class Christmas
             extends Holiday
             implements Observable, Ignorable
{
   // code here
}
```



Christmas API

Holiday API

Observable API

Ignorable API

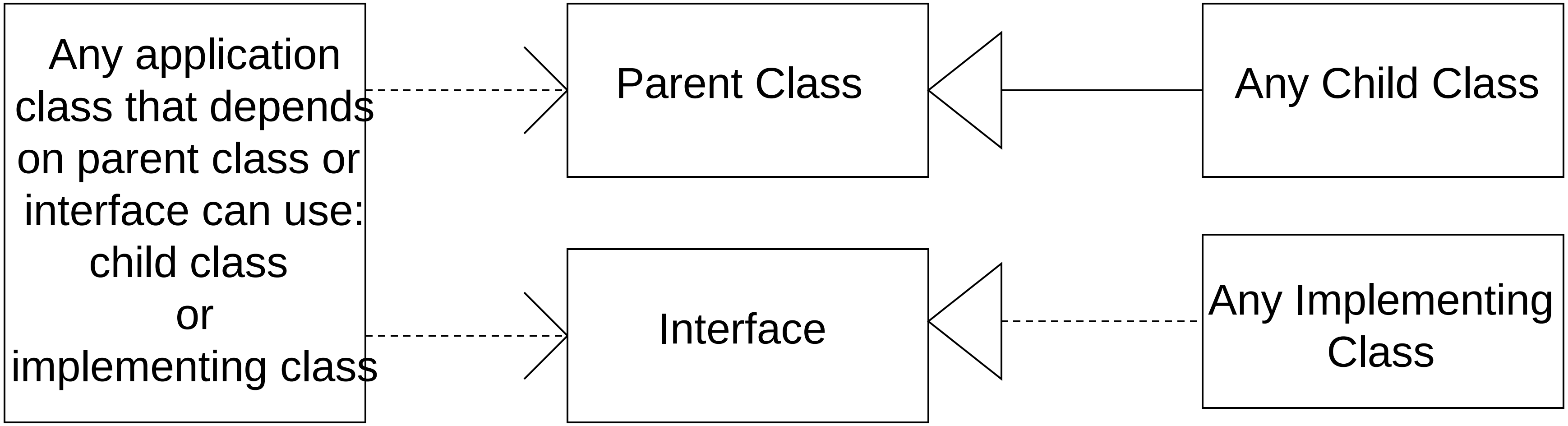
Object of class Christmas

# Summary of Polymorphism

| **Object <u>instantiated</u> as:** | **Can/Cannot be cast:** |
|---|---|
| Child or Later Descendent Class | 🟩 To Parent or Earlier Ancestor (and back to its original class) |
| Parent or Earlier Ancestor Class | 🟥 To Child or Later Descendent |
| Implementing Class | 🟩 To any Interface it implements (and back to its original class) |
| | 🟥 To any “incompatible class” |
| 🟥 Any Abstract Class or Interface cannot be instantiated | |

# Polymorphism: UML Class Diagrams

- You see how both Inheritance and Interfaces can be used to support polymorphic object references
- You should now be able to understand why both Inheritance and Interfaces are shown with the same "generalization" arrow icon in UML class diagrams



Any application class that depends on parent class or interface can use: child class or implementing class
Parent Class
Any Child Class
Interface
Any Implementing Class